July 2005

Run-Time Checking of Dynamic Properties

Oleg Sokolsky
University of Pennsylvania, sokolsky@cis.upenn.edu

Usa Sammapun
University of Pennsylvania, usa@cis.upenn.edu

Insup Lee
University of Pennsylvania, lee@cis.upenn.edu

Jesung Kim
University of Pennsylvania, jesung@cis.upenn.edu

Recommended Citation

Oleg Sokolsky, Usa Sammapun, Insup Lee, and Jesung Kim, "Run-Time Checking of Dynamic Properties", . July 2005.

Postrpint version. *Proceedings of the Fifth Workshop on Runtime Verification (RV'05)*
Note: This is a preliminary version. The final version will be published in *Electronic Notes in Theoretical Computer Science*
Publisher URL: www.elsevier.nl/locate/entcs

Run-Time Checking of Dynamic Properties

Abstract
We consider a first-order property specification language for run-time monitoring of dynamic systems. The language is based on a linear-time temporal logic and offers two kinds of quantifiers to bind free variables in a formula. One kind contains the usual first-order quantifiers that provide for replication of properties for dynamically created and destroyed objects in the system. The other kind, called attribute quantifiers, is used to check dynamically changing values within the same object. We show that expressions in this language can be eficiently checked over an execution trace of a system.

Comments
Postrpint version. *Proceedings of the Fifth Workshop on Runtime Verification (RV'05)*
Note: This is a preliminary version. The final version will be published in *Electronic Notes in Theoretical Computer Science*
Publisher URL: www.elsevier.nl/locate/entcs

RV 2005 Preliminary Version

Run-Time Checking of Dynamic Properties [1]

Oleg Sokolsky, Usa Sammapun, Insup Lee, and Jesung Kim

Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104-6389
U.S.A.
{*sokolsky,usa,lee,jesung*}*@cis.upenn.edu*

Abstract

We consider a first-order property specification language for run-time monitoring of dynamic systems. The language is based on a linear-time temporal logic and offers two kinds of quantifiers to bind free variables in a formula. One kind contains the usual first-order quantifiers that provide for replication of properties for dynamically created and destroyed objects in the system. The other kind, called attribute quantifiers, is used to check dynamically changing values within the same object. We show that expressions in this language can be efficiently checked over an execution trace of a system.

1 Introduction

Run-time verification is a novel light-weight verification technique that applies formal analysis techniques directly to the executing system rather than its model. Analysis of a formally specified property is performed with respect to a given execution trace of the system. Algorithms introduced in [8] showed that analysis of commonly used propositional temporal logics such as LTL can be performed using space that is independent of the length of the execution trace. These complexity results compare favorably to model checking, which is PSPACE-complete for LTL [12]. This often makes model checking of large systems impractical, while run-time verification is not constrained by the size of the system. In this work, we show that benefits of run-time verification can be extended to first-order specification languages that are much harder – and often undecidable – to check over models.

In our prior work [10], we have developed LC_p, a propositional temporal logic of events and conditions that is suitable for efficient run-time verification

[1] This research has been supported in part by DARPA PCA F33615-00-C-1887, NSF CCR-0209024, ARO DAAD19-01-1-0473.

of temporal properties. LC_p formulas are interpreted over a trace of observations of a program execution. The trace contains occurrences of observable *events* that happen during the execution. These primitive events serve as atomic propositions of the logic and are used to define more complex events and conditions that express system requirements and their violations.

In this paper, we extend LC_p to a first-order temporal logic LC_ν that captures two aspects of run-time monitoring that were not handled in LC_p: event attributes and dynamic indexing of properties.

Event attributes. Events, occurring during an execution of the system, usually carry additional information that is important for expressing and checking system properties. If a system variable changes its value, we may need to know what new value is assigned; if a system call happens, the values of the arguments of the call may be important. We may also want to record the time instance when an event occurred and use it to check timing properties. We call the values that are associated with an event occurrence *event attributes*. Event attributes may be used to define new conditions in property specifications. When a new instance of an event occurs, the new value of the attribute affects evaluation of the property. While some support for handling of event attributes was provided in MEDL [11], a specification language based on LC_p, precise semantics for it were never given.

Dynamic indexing of properties. Propositional temporal logics used in verification check fixed collections of properties. Consider the (in)famous railroad crossing example, where we check that whenever a train is in crossing, announced by an event *trainCrossing*, the gate is closed, that is, we observed an event *GateDown* and there has been no event *GateUp* since. If we have multiple crossings that we need to verify simultaneously with respect to the same set of properties, we have individual events $GateUp_i$, $GateDown_i$, etc., for each crossing i. Since crossings are rarely added and even more rarely removed, it makes sense to assume that the set of crossings is fixed. Every property is written by creating a copy of a property indexed by the identity of the crossing. Because properties are duplicated, checking is naturally performed independently for each property. The run-time state needed by the checker to evaluate each property is created statically, separately for each property. This is easy to do since we know which crossings are there and properties for any other crossing simply do not exist, i.e., undefined. A stray event *GateUp* for a non-existent crossing may cause an error message by the checker or, more likely, will simply be ignored like any other unexpected event.

The situation changes when, instead of railroad crossings, we want to check properties of trains. Intuitively, the situation is the same: we have a set of properties per train that can be checked individually. However we do not know which trains are there as trains come and go. The set of properties to check can be still thought as indexed by the identity of the train; however, now this index appears as a free variable and we are forced to enter the realm of first-order temporal logics, from the specification language perspective. From the checking perspective, the checker needs to dynamically identify what set of properties need to be checked, allocate and – to remain efficient – deallocate space for the run-time state, and reject events that do not correspond to properties being currently checked.

In the logic LC_ν, some of event attributes are treated as index values rather than data points. We use variables to represent indices as well as data attributes. During an execution, the variables are bound to values that are supplied by occurrences of events in the trace. However, the way this binding occurs is different for indices and event attributes. Consider checking the property that the speed of a train is within a fixed speed limit. Assume that speed readings are manifested in the execution trace as events $\mathit{speed}(i, v)$, with i being the identity of the train and v giving the current velocity of the train i. Suppose for now that we have only one train i_1. The condition $v \leq \mathit{limit}$ is evaluated with respect to the value of v supplied by the last occurrence of $\mathit{speed}(i_1, v_1)@t$. All previous occurrences of the same event $\mathit{speed}(i_1, v_2)@t'$ with $t' < t$ become irrelevant. However, suppose now we have another train i_2. In this case, an occurrence $\mathit{speed}(i_2, v_2)@t'$ should not, of course, be overshadowed by the later occurrence of $\mathit{speed}(i_1, v_1)@t$. If we now assume that arrival of new trains within the visibility range of the monitor is announced by an event $\mathit{newTrain}(i)$, we see that the values brought as attributes of the $\mathit{newTrain}$ event should be bound differently from the values brought as attributes of the speed event.

Accordingly, we introduce two different kinds of quantifiers that bind event attributes, depending on whether they are to be used as index values or data. Index values can be quantified using the usual first-order quantifiers. Since the contents of index sets can be changed dynamically, each quantifier is decorated with events that add and remove values from the index set. Other event attributes are bound by a special quantifier, which we call an *attribute quantifier*. Attribute quantifiers are reminiscent of freeze quantifiers used in real-time temporal logics [2,9] and logics for object-based systems [5], as well as assignment quantifiers [13] used in logics for temporal databases. There is a difference between attribute quantifiers and the common use of freeze quantifiers. Normally, freeze quantifiers bind the value of a variable to a certain value in the environment at a specific point in time. However, in our case, the only connection to the environment, that is, the system being monitored, is via primitive events and conditions. Therefore, with every attribute quantifier, we specify the primitive event that brings the value as an attribute. The quantifier, then, binds a variable in the formula to the attribute of the most recent occurrence of the specified primitive event.

Related work. The closest point of reference is provided by formalisms aimed at describing the evolution of systems with object creation and destruction. The *allocation temporal logic AllTL* [7] for object-oriented systems is designed specifically for describing patterns of object creation and destruction. A similar logic with different semantics, *evolution temporal logic* ETL [14], is used to describe temporal aspects of heap allocation. Most other temporal logics for object-oriented systems, such as BOTL [6], do not consider dynamic object creation and thus do not have to offer dynamic indexing in the language.

Checking with respect to general first-order temporal logics are known to be undecidable [1]. Many variants of first-order temporal logics allow semi-decidable procedures and are implemented in a variety of theorem provers. For example, STEP [4] uses first-order LTL to express properties of the systems.

A distinctive feature of LC_ν is that it aims at run-time verification rather than model checking. Model checking algorithms operate directly on the system model and thus the semantics for the logic can be defined directly in terms of the system execution. In the case of run-time verification, the checker has access to the system execution only via a sequence of observations. Because of this, constructs of LC_ν are tied to events, which bring the values changed during the execution. This approach to the semantic definition allows us to achieve clean separation between *trace extraction*, which concerns system instrumentation to produce the necessary observations, and *trace checking*, which operates on the extracted trace. This separation provides for property specifications that are independent of a particular system implementation. These properties can then be used to check any implementation of the system by a generic checker, after applying an implementation-specific extraction.

A very expressive logic for run-time verification EAGLE has been introduced in [3]. The use of parameters in EAGLE expressions provides for indexing of properties and quantification. However, there is no distinction between creation/destruction events that change the index sets and ordinary events used in the checking. Because of this, we believe that properties expressed in LC_ν will be easier to understand and check.

2 Logic for Dynamic Events and Conditions

Intuition for the logic. We specify properties of systems by means of *events* and *conditions*. Conditions are statements about the current state of the system, which retain their value until the state changes. By contrast, events are instantaneous observable state changes that change the value of some conditions. Attributes of events specify values that are associated with a state change. For a software system, conditions may be expressions over the system variables and location of control, while primitive events may be assignments to these variables and changes in control locations such as function calls. In that case, attributes of events can be values assigned to a variable or parameters of a call. More complex events and conditions can be constructed by the operators of the logic. Ultimately, formulas of the logic define invariants (properties that should always be true during an execution) and alarms (events that notify the user of abnormal behaviors).

Syntax. We assume a countable set of variables $\mathcal{X} = \{X_1, X_2, \ldots\}$. Variables may assume values taken from some domain $\mathcal{V}$. We also assume a set of functional symbols that represent primitive events $\mathcal{E}_p = \{e_1, e_2, \ldots\}$, each symbol having a fixed arity. We will use $\bar{X}$ to denote a vector $X_1, \ldots, X_n$ and, when writing terms $e(\bar{X})$ or $p(\bar{X})$ will always assume that the size of the vector corresponds to the arity of the function symbol.

A *predicate* $p(\bar{X})$ is a function $\bar{X} \to \{false, true\}$. The exact definition of $p(\bar{X})$ is unimportant, as long as we assume that it can be computed in $O(\bar{X})$ time. In examples, predicates will be constructed using relational operators applied to arithmetic expressions over from $\bar{X}$ and a special function symbol $time(e)$. The latter is understood as the time of the most recent occurrence of the primitive event e.

An occurrence of an event $e(X_1, \ldots, X_n)$ is a ground term $e(v_1, \ldots, v_n)$, i.e., an assignment of values to the variables of the event, which occurs at some time instance t. We will refer to these values as the *attributes* of the event. We denote the set of primitive event occurrences as $\mathcal{E}_g$.

The logic has two sorts: conditions and events. The set of conditions is denoted by $\mathcal{C}$, while the set of events is denoted by $\mathcal{E}$. We will use C (possibly primed or subscripted) to range over $\mathcal{C}$ and E to range over $\mathcal{E}$. Lowercase e are used to mean primitive events. We will use f to range over formulas in $\mathcal{C} \cup \mathcal{E}$. Figure 1 shows the syntax of conditions and events. For conditions, we use operators $\wedge$ and $\Rightarrow$ as the commonly defined abbreviations. For events, for which there is no negation operator defined, conjunction is a primary operator.

$$\begin{array}{lcl}
\langle C \rangle & ::= & p(\bar{X}) \mid \texttt{defined}(\langle C \rangle) \mid [\langle E \rangle, \langle E \rangle) \mid \neg \langle C \rangle \mid \langle C \rangle \vee \langle C \rangle \\
& & \mid \exists x[e_1, e_2].\langle C \rangle \mid \forall x[e_1, e_2].\langle C \rangle \mid x@e(\bar{X}).\langle C \rangle \\
\langle E \rangle & ::= & e(\bar{X}) \mid \texttt{start}(\langle C \rangle) \mid \texttt{end}(\langle C \rangle) \mid \langle E \rangle \wedge \langle E \rangle \mid \langle E \rangle \vee \langle E \rangle \mid \langle E \rangle \ \texttt{when} \ \langle C \rangle \\
& & \mid \exists x[e_1, e_2].\langle E \rangle \mid \forall x[e_1, e_2].\langle E \rangle
\end{array}$$

Fig. 1. The syntax of conditions and events

The intuition behind the first-order quantifiers used to define events and conditions is that each occurrence of the event e_1 introduces a new "checking instance", indexed by the value of the attribute x in the e_1 occurrence. When event $e_2(x)$ occurs, the checking occurrence that corresponds to the value x is removed. We refer to e_1 as the creation event and to e_2 as the destruction event. Quantifiers $\exists$ and $\forall$ allow us to make statements about collections of these checking occurrences. The intuition for the attribute quantifier @ allows us to use the value of an event attribute for evaluation of predicates.

"Index attributes" and "value attributes" are conceptually different entities. Rather than introduce separate types for these attributes, we impose a syntactic well-formedness constraint, which requires that if a variable is bound by a first-order quantifier, it can only be used in an equality predicate. We also assume that in all quantifier formulas, $\exists x[e_1(\bar{X}), e_2(\bar{X})].f$, $\forall x[e_1(\bar{X}), e_2(\bar{X})].f$, or $x@e(\bar{X}).f$, $x \in \bar{X}$.

The set of *free variables* of a formula (event or condition) f, $fv(f)$ is given recursively on the structure of the formula in the usual way. Important clauses are $fv(e(\bar{X})) = fv(p(\bar{X})) = \{\bar{X}\}$, $fv(\forall x[e_1, e_2].f) = fv(\exists x[e_1, e_2].f) = fv(x@e(\bar{X}).f) = fv(f) \backslash \{x\}$. (Operator $\backslash$ is the set difference).

Semantics. Intuitively, we think of events as an abstract representation of time and conditions as abstract representation of data. That is, conditions represent the relevant state of the system or of the checker, while events evaluate to the time instances when "interesting" changes occur.

A model M is a tuple $\langle \mathcal{T}, S, \tau, L_E \rangle$, where $\mathcal{T}$ is a time domain (which could be integers, rationals, or reals), $S = \{s_0, s_1, \ldots\}$ is a finite or infinite sequence of states, $\tau : S \to \mathcal{T}$ is a timestamping function, and L_E is a relation over $S \times \mathcal{E}_g$ that tells which primitive events occur in the states of S. That is, in each state s, $(s, e) \in L_E$ for each event occurrence of e at s. By $L_E(s)$ we will denote the set of event occurrences $\{e_i\}$ so that $(s, e_i) \in L_E$. The mapping τ defines the time at each state, and it satisfies the requirement that $\tau(s_i) < \tau(s_j)$ for all $i < j$, i.e., the time at a later state is greater. Intuitively, the states represent *observations* delivered to us from the system, and the order of states is determined by the observation order. The requirements on τ reflect the assumptions: observations are delivered in the order they were produced by the system, and the clock granularity is fine enough to assign a different timestamp to each observation.

The meaning of a formula with free variables is given up to two partial valuation functions $V, I : \mathcal{X} \to \mathcal{V}$, which assign values to the free variables of the formula, to which we refer as the *value* and *index assignments*, respectively. The intuition for having two valuation functions is to be able distinguish between "index attributes" and "value attributes" during evaluation. Valuation functions V, I are called compatible, denoted $V \asymp I$, if their domains are disjoint, that is, $V(x) \neq \perp$ implies $I(x) = \perp$ and vice versa. Given a primitive event $e(\bar{X})$ and compatible valuations V, I, $e(\bar{X})[V, I]$ will denote the event occurrence $e(\bar{v})$, if for each i, either $V(X_i) = v_i$ or $I(X_i) = v_i$. Similar notation is used for predicates. If neither V nor I assign value to some variable X, $p(\bar{X})[V, I]$ denotes $\perp$. Given a valuation V, $V\{X_i \mapsto v_i\}$ is a valuation that has $V(X_i) = v_i$ for every i and agrees with V on all other variables.

In order to define what we mean by a condition C being true in model M at time t ($M, t, V, I \models C$), we need to define what we mean by its denotation ($\mathcal{D}^t_{M,V,I}(C)$). This is defined in Figure 2. Using this we define the meaning of $M, t, V, I \models C$, and of an event E occurring in a model M at time t ($M, t, V, I \models E$). The formal definition is given in Figure 3.

We interpret conditions over three values, $true$, $false$, and $\perp$ (undefined). A predicate $p(\bar{X})$ is computed according to the given valuations of its free variables and is undefined if the valuation of any of the free variables is undefined. The predicate defined(C) is true whenever the condition C has a value $true$ or $false$. Negation ($\neg C$) and disjunction ($C_1 \vee C_2$) are interpreted classically whenever C, C_1 and C_2 have boolean values $true$ or $false$; $\neg \perp = \perp$, $\perp \vee true = true$, and $\perp \vee false = \perp$.

$$
\begin{array}{lcl}
\mathcal{D}^t_{M,V,I}(p(\bar{X})) & = & p(\bar{X})[V,I] \\
\mathcal{D}^t_{M,V,I}(\text{defined}(C)) & = & \begin{cases} \mathit{true} & \text{if } \mathcal{D}^t_{M,V,I}(C) \neq \perp \\ \mathit{false} & \text{otherwise} \end{cases} \\
\mathcal{D}^t_{M,V,I}([E_1, E_2)) & = & \begin{cases} \mathit{true} & \text{if there exists } t_0 \leq t \text{ such that} M, t_0, V, I \models E_1 \\ & \quad \text{and for all } t_0 \leq t' \leq t, M, t', V, I \not\models E_2 \\ \mathit{false} & \text{otherwise} \end{cases} \\
\mathcal{D}^t_{M,V,I}(\neg C) & = & \begin{cases} \mathit{true} & \text{if } \mathcal{D}^t_{M,V,I}(C) = \mathit{false} \\ \perp & \text{if } \mathcal{D}^t_{M,V,I}(C) = \perp \\ \mathit{false} & \text{if } \mathcal{D}^t_{M,V,I}(C) = \mathit{true} \end{cases} \\
\mathcal{D}^t_{M,V,I}(C_1 \vee C_2) & = & \begin{cases} \mathit{true} & \text{if } \mathcal{D}^t_{M,V,I}(C_1) \text{ or } \mathcal{D}^t_{M,V,I}(C_2) \text{ is } \mathit{true} \\ \mathit{false} & \text{if } \mathcal{D}^t_{M}(C_1) = \mathcal{D}^t_{M,V,I}(C_2) = \mathit{false} \\ \perp & \text{otherwise} \end{cases}
\end{array}
$$

Fig. 2. Denotation for conditions

For primitive events, the meaning is given to the event occurrence obtained by the valuations V and I, and is defined by the labels on the states. Conjunction $(E_1 \wedge E_2)$ and disjunction $(E_1 \vee E_2)$ defined classically in the temporal domain; so $E_1 \wedge E_2$ is present only when both E_1 and E_2 are present, whereas $E_1 \vee E_2$ is present when either E_1 or E_2 is present.

There are some natural events associated with conditions, namely, the instant when the condition becomes *true* (start(C)), and the instant when the condition becomes *false* (end(C)). Notice, that the event corresponding to the instant when the condition becomes $\perp$ can be described as end(defined(C)). Also, any pair of events define an interval of time, so forms a condition $[E_1, E_2)$ that is *true* from event E_1 *until* event E_2. The event (E when C) is present if E occurs at a time when condition C is *true*.

Finally, the meaning to formulas with quantifiers is given in the usual way by transforming the assignments that are used to evaluate the formula. For the first-order quantifiers, an occurrence of e_1 with the value v_i for the attribute x_i makes the checker to evaluate f in every state of the trace in the index valuation containing v_i, until a matching occurrence of e_2 is found. By contrast, the attribute quantifier $x@e(\bar{X})f$ makes the checker evaluate f with the value assignment containing the value of x_i from the last occurrence of e.

As usual, formulas are interpreted up to renaming of bound variables. Clearly, the formula

$$\forall x[e_1(x), e_2(x)].f(x) \wedge \forall x[e'_1(x), e'_2(x)].f'(x)$$

Semantics of conditions:		
$M, t, V, I \models C$	iff	$\mathcal{D}^t_{M,V,I}(C) = true$
Semantics of events:		
$M, t, V, I \models e(\bar{X})$	iff	there exists state s_i such that $\tau(s_i) = t$
		and $(s_i, e(\bar{X})[V, I]) \in L_E$.
$M, t, V, I \models \text{start}(C)$	iff	$\exists s_i$ such that $\tau(s_i) = t$ and $M, \tau(s_i), V, I \models C$
		and if $i > 0$ then $M, \tau(s_{i-1}), V, I \not\models C$.
$M, t, V, I \models \text{end}(C)$	iff	$\exists s_i$ such that $\tau(s_i) = t$ and $M, \tau(s_i), V, I \models \neg C$
		and if $i > 0$ then $M, \tau(s_{i-1}), V, I \not\models \neg C$.
$M, t, V, I \models E_1 \vee E_2$	iff	$M, t, V, I \models E_1$ or $M, t, V, I \models E_2$.
$M, t, V, I \models E_1 \wedge E_2$	iff	$M, t, V, I \models E_1$ and $M, t, V, I \models E_2$.
$M, t, V, I \models E$ when C	iff	$M, t, V, I \models E$ and $M, t, V, I \models C$;
Semantics of quantifiers:		
$M, t, V, I \models \exists x_i[e_1, e_2].f$	iff	$\exists s$ with $\tau(s) \leq t$. $\exists v_i \in \mathcal{V}$. $\exists V' \asymp I$ such that
		$M, \tau(s), V', I\{x_i \mapsto v_i\} \models e_1$, $\forall \tau(s) \leq t' < t. \forall V'' \asymp I$
		$M, t', V'', I\{x_i \mapsto v_i\} \not\models e_2$, and $M, t, V, I\{x_i \mapsto v_i\} \models f$.
$M, t, V, I \models \forall x_i[e_1, e_2].f$	iff	$\forall s$ with $\tau(s) \leq t$, ($\exists v_i \in \mathcal{V}$. $\exists V' \asymp I$ such that
		$M, \tau(s), V', I\{x_i \mapsto v_i\} \models e_1$, $\forall \tau(s) \leq t' < t. \forall V'' \asymp I$.
		$M, t', V'', I\{x_i \mapsto v_i\} \not\models e_2) \Rightarrow M, t, V, I\{x_i \mapsto v_i\} \models f$.
$M, t, V, I \models x_i@e.f$	iff	$\exists s$ with $\tau(s) \leq t$, $\exists v_i \in \mathcal{V}$. $M, \tau(s), V\{x_i \mapsto v_i\}, I \models e$,
		such that $(\forall \tau(s) \leq t' < t, v \neq v_i. M, t', V\{x_i \mapsto v\}, I \not\models e)$
		and $M, t, V\{x_i \mapsto v_i\}, I \models f$.

Fig. 3. Semantics of events and conditions.

is equivalent to

$$\forall x[e_1(x), e_2(x)].f(x) \wedge \forall y[e_1'(y), e_2'(y)].f'(y).$$

However, events that appear in quantifiers can restrict the effect of such renaming. This is especially clear in the case of attribute quantifiers. Consider $x@e_1(x).f_1(x) \wedge y@e_1(y).f_2(y)$. Even though x and y are bound by different quantifiers, they are not independent and refer to the same attribute of the same event. That is, in any evaluation, x and y will have the same value.

3 Examples: Dynamic Real-Time Systems

We sketch two examples taken from the domain of real-time systems. A typical real-time system consists of a collection of tasks. Tasks can be added upon user requests or in response to the changing environment. Tasks execute concurrently on the same processor. A task is *dispatched* to perform a certain computation, either periodically or in response to some event. Once the task is dispatched, its computation should complete by a given *deadline*. In *hard* real-time systems, such as high-frequency controllers, a missed deadline is considered to be a catastrophic event.

Suppose that changes to the set of tasks are announced by primitive events $addTask(id)$ and $removeTask(id)$. Different invocations of a task may carry different deadlines. A task dispatch is modeled by an event $dispatch(id, d)$, where d is the deadline for the task. When the task completes its execution, an event $complete(id)$ occurs. The property that every deadline is met on each task dispatch is expressed as

$$\begin{aligned}&\forall i[addTask(i), removeTask(i)].\\ &\qquad d@dispatch(i,d).time(complete(i)) - time(dispatch(i,d)) \leq d. \quad (1)\end{aligned}$$

On the other hand, *soft* real-time systems, such as multimedia streaming servers, do not aim to make every deadline. Instead, the goal is to provide a certain level of quality of service (QoS), for example measured by the rate at which the server produces stream data. The rate can be dynamically adjusted based on the network conditions. A scheduler controls execution of tasks, trying to establish the target rate. User requests for streams are categorized into service classes, and each class is characterized by fixed QoS bounds. The property we check in this case is that rate adjustments obey the bounds of the service class. Formula (2) checks that a rate bound is never exceeded. It assumes the following events: $addClass(sc, b)$ that introduces a service class sc with the associated rate bound b and a matching event $removeClass$; $addStream(m, sc)$ announces the creation of a new stream m in the service class sc, and the matching completion event $closeStream$; finally $setRate(m, r)$ adjusts the target rate r for the stream m.

$$\begin{aligned}&\forall sc[addClass(sc,b), removeClass(sc)].b@addClass(sc,b).\\ &\qquad \forall m[addStream(m,sc), closeStream(m)].r@setRate(m,r).r < b \quad (2)\end{aligned}$$

4 Checking LC_ν

In this section, we present an algorithm for on-line checking of a LC_ν with respect to a continuously unfolding trace of an execution of the monitored system. We begin by briefly reviewing the checking algorithm for LC_p from [10]. Then we discuss, using the examples from Section 3, the necessary extensions to deal with event attributes and dynamic indexing. Finally, we present the checking algorithm.

4.1 Checking LC_p

The syntax of an LC_p formula is given by the grammar of Figure 1, without any quantifiers, and with predicates built using only the function symbol *time*. The arity of every primitive event is assumed to be zero. Thus, LC_p is the propositional subset of LC_ν.

For a given LC_p formula f, the checker maintains an acyclic graph that represents the dependencies between subformulas in the formula. Each node in the graph corresponds to subformula. A node representing a condition stores the current value of the condition, and has a temporary variable *new* used in the evaluation. A node representing an event stores the time of the most recent occurrence of that event. Since there is a one-to-one correspondence between nodes in the graph and subformulas of f, we will use f to denote the node in the graph as well as the formula, as long as the use is clear from the context. There is an edge $f_1 \rightarrow f_2$ in the graph if f_1 is an immediate subformula of f_2. Nodes that correspond to primitive events are, therefore, the leaves of the graph. A predicate node p that uses $time(E)$ function has the incoming edge $E \rightarrow p$. Each node f in the graph is assigned a height, denoted $h(f)$, defined as the number of steps in the longest path from a leaf in the graph to the node. For each height $h \neq 0$, there is a set A_h of nodes that recently changed their values; initially all sets are empty.

The direction of edges reflects the order of evaluation of nodes by the checker, which is done in the bottom-up fashion, driven by occurrences of events in the trace. Given a model $M = \langle T, S, \tau, L_E \rangle$, the checker evaluates a given formula f for each state $s \in S$, in the order of states in S, as follows: First, values of primitive events are processed based on the information contained in s. If $(s, e) \in L_E$ for some primitive event e, the node that corresponds to e is updated to record $\tau(s)$ and every node f with $e \rightarrow f$ is added to $A_{h(f)}$. Then, the sets A_h are processed in the order of increasing height. Each node in the current list is re-evaluated using the new values of its predecessor nodes, according to the semantics of the operator. Predicate nodes are evaluated using the time values stored in the event nodes. If the value of the current node changes, its successors are added to their respective change sets and the node is removed from its change set. When all changes are processed, the values of condition nodes are set to their *new* values, and the algorithm considers the next state in the trace.

As an example, consider a propositional version of property (1). Assume we have a single task with a fixed deadline, say, $d = 5$. The alarm that we want the checker to raise is $end(time(complete) - time(dispatch) \leq 5)$. Figure 4,a illustrates the graph for this formula. The figure shows the values of the nodes in the graph for the trace $dispatch@2, complete@8$. At this point, the values

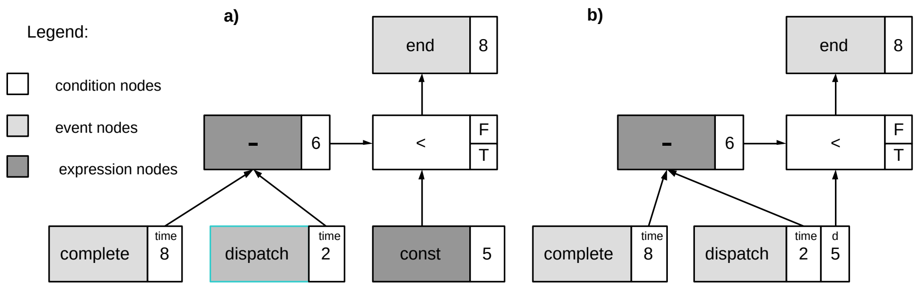

Fig. 4. A propositional formula

of the predicate condition changes from true to false (the new value is shown in the top-right corner and the old value in the bottom-right corner).

4.2 Checking formulas with event attributes

As the first step towards the checking algorithm for LC_ν, we discuss the subset of LC_ν, in which all variables are bound by attribute quantifiers. We extend the evaluation graph data structure to incorporate event attributes and predicates. A node that represents a primitive event is extended with an array of variables storing the attributes of the event. When the event occurs, the values of the attributes are assigned to the respective variables in the array and all successors of the event node are set for re-evaluation.

A predicate is represented by a subgraph that represents the structure of the expressions that form the predicate. The leaf nodes in this subgraph correspond to variables and to *time* functions. In the graph, these leaf nodes are replaced by references to the values stored in an event node. That is, there is an edge $e \rightarrow p$ whenever p uses $time(e)$. There is an edge $e \rightarrow p$ if $x \in fv(p)$ and p is within the scope of $x@e(\bar{X})$. We pictorially represent these edges as originating from the right value. Figure 4,b shows the graph for the formula $d@dispatch(d).end(time(complete) - time(dispatch(d)) \leq d)$.

It remains to handle the case when a primitive event is within the scope of an attribute quantifier for a different primitive event (for example, $x@e_1(x).e_2(x)$). Such formulas are syntactically transformed into an equivalent formula by adding new attribute quantifiers and renaming the bound variables. In the example, the equivalent formula is $x@e_1(x).y@e_2(y).(e_2(x) \text{when} x = y)$. That is, an occurrence of e_2 affects evaluation of the formula if the attributes match.

The checking algorithm itself does not change from the case of LC_p.

4.3 Checking full LC_ν

The main difference in handling the first-order quantifiers of LC_ν is that the space needed to check the property can increase and decrease dynamically. The algorithm will still operate on a graph data structure similar to the preceding sections, but data stored in the graph nodes would be different. Instead of storing a single boolean value for conditions or a time stamp for events,

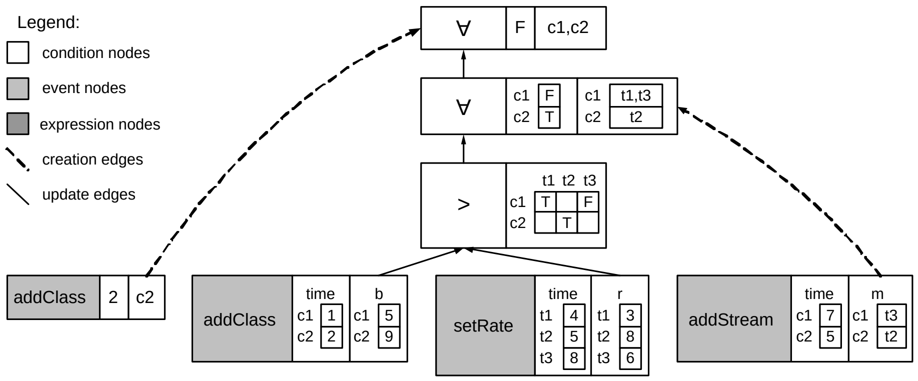

Fig. 5. A first-order formula

the annotation of a node would be a map associating valuations of the free variables in the respective subformula to a boolean value or a timestamp, as appropriate for the node type.

Graphs for LC_ν formulas contain a new kind of graph node, which corresponds to subformulas, in which the main operator is a first-order quantifier that binds a variable x. Such a node f contains the same value map that a regular node has. In addition, the node is associated with a map that, for each valuation of the free variables of the node, gives the current set of values for x. This set contains the values taken from those occurrences of the construction event, for which there have been no matching destruction event. We will refer to such a set as the index set for x. Index sets will determine the domain of the value mappings in the nodes for the subformulas of f that have x free.

Consider property (2). The graph in Figure 5 shows value maps stored in the nodes when the violation is found after the evaluation of the trace

$$addClass(c1, 5)@1, addClass(c2, 9)@2, addStream(t1, c1)@3, setRate(t1, 3)@4,$$
$$addStream(t2, c2)@5, setRate(t2, 8)@5, addStream(t3, c1)@7, setRate(t3, 6)@8$$

The graph has two quantifier nodes. The first node does not have free variables and thus contains all service class identifiers that are brought by *addClass* events. The other quantifier node has the variable sc free, and thus keeps the set of stream identifiers separately for each service class. Its child, the relational node, has both sc and m free, and thus its value mapping is the function of two arguments. Note, however, that some combinations yield undefined values, because each stream belongs to only one service class. Nodes that correspond to primitive events store the time of last occurrence and values of attributes, quantified by attribute quantifiers, for each combination of values of attributes quantified by first-order quantifiers. Note that the same primitive event can be used in different contexts in the formula. That is, an attribute can be quantified by an attribute quantifier in one subformula and by a first-order quantifier in another subformula. Such situation, represented in the example by the event *addClass*, requires us to keep a set of nodes for the same primitive event e, denoted $nodes_e$.

The graph also has additional edges that capture creation and destruction of index set elements. Creation and destruction edges are handled similar to the attribute edges, except that each new attribute value is added to an index set in the target node, rather than trigger the re-evaluation of the target node. Destruction edges are not shown in Figure 5 to avoid clutter. All edges other than creation or destruction edges are referred to as update edges.

During evaluation, nodes f that require re-evaluation are recorded in the sets A_h, separately for each height. To record what valuation I of the free variables needs to be used in re-evaluation, each set A_h contains pairs (f, I).

To illustrate the evaluation, consider arrivals of the events $setRate(t1, 3)@4$, $addStream(t2, c2)@5$, and $setRate(t2, 8)@5$. Note that the last two events arrive simultaneously. In the first case, m is a free variable of the node for $setRate$ and r is the attribute. Therefore, the entry for $t1$ is updated in the primitive event node to record the time of arrival and the new value of r, which becomes 3. The parent of the node, which is the relational node with height 1, is marked for re-evaluation of all values in the mapping that corresponds to the $t1$ column. The value for $(c2, t1)$ is undefined, and the value for $(c1, t1)$ becomes true. When the second event arrives, sc is the free variable of the primitive event node, and the new value for m and the time of arrival is recorded for $sc = c2$. Following the creation edge, $t2$ is added to the index set for $c2$ in the quantifier node, which, in turn, makes entries for $t2$ in the node that have m as the free variable, that is, the relational node and the node for $setRate$. Then, the last event is processed as described above. The presence of this last event is important to demonstrate that creation events need to be processed before any other events in the same state, otherwise the value map may not yet have the entries to store the new values. Similarly, destruction events need to be processed after all regular updates are performed.

4.4 Checking algorithm

After giving examples to illustrate different aspects of checking, we are ready to present the checking algorithm. The algorithm consists of the static and dynamic phases. The static phase algorithm performs analysis of the formula structure and produces the graph used in the dynamic phase. The dynamic phase processes the states of the execution trace one by one and performs the evaluation on the graph, adding entries into value maps in the nodes as creation events arrive and removing them as in response to destruction events.

Notation. We will use the following notation to describe the algorithm. When a node f is using a specific attribute i of a primitive event e, we reflect this as an edge label: $e \xrightarrow{i} f$. In an event occurrence $e(\bar{v})$, the values of its index attributes define a valuation I_e. Given a valuation I, we often have to extend its domain to a larger set of variables. We can define multiple extensions that agree with I, by taking different values for the new variables. Given a node f

and a valuation I, the set of possible extensions of I to the free variables of f is denoted $\mathcal{I}_f(I)$. For a quantifier node f, $nodes_f$ will denote the set of nodes that contain the free variable bound by f. $Ind_I(f)$ will denote the index set for a valuation I. By $compute_I(f)$, we will denote the function that updates the value of a node from the value of its subnodes, with a given valuation of its free variables I. In other words, $compute_I(f)$ is the semantic function of the subformula f. Finally, val_f is the value map stored in the node f.

The static phase. Construction of the graph proceeds by starting at the main operator of a given formula and proceeding recursively in a depth-first manner through its subformulas. On the way down into the tree of subformulas, information about bindings is passed to the nodes on the lower level. Reaching a leaf of the subformula tree, the algorithm determines, (1) if the node is a primitive event, which of the attributes are to be treated as index values and which are to be stored as attributes; (2) if the node is a variable in a predicate, which event attribute it is bound to. On the way up, the sets $nodes_f$ for each quantifier node f are computed, and heights are assigned to every node in the graph.

Initialization of nodes. Variables in the nodes that represent values of the respective subformula need to be initialized as the node is created, or dynamically as the valuation map is extended by adding a new element to an index set. The rules for initialization are as follows: event values are initialized to $\perp$, since the event has never happened. Condition nodes are evaluated according to $compute_I(f)$. Initialization is performed in the topological order so that all predecessors of a node have been already initialized.

The dynamic phase of the checking is given by Algorithm 1, which is invoked on a given execution trace $M = \langle \mathcal{T}, S, \tau, L_E \rangle$. The algorithm processes each state $s \in S$ in the trace once and in the order the states appear in the trace. Any information that needs to be kept after a state is processed is stored in the variables associated with nodes in the graph. Because of this, the processed prefix of the execution trace need not be stored.

When a new state of the trace arrives, the algorithm processes each event occurrence e in the state as follows. First, each node immediately affected by e is updated with the values of the attributes of the occurrence. Then, creation edges of the node are followed to add new index values to index sets that depend on index attributes of e. After that, update edges are followed to add the potentially affected nodes to the update sets at the appropriate heights.

Once all event occurrences are processed, the sets A_h are processed in the order of increasing height h. At this stage, all index sets are fixed and processing follows the pattern of the LC_p checking algorithm for specific index values, except that destruction edges are used to mark certain index values for removal. The actual removal of index values is performed at the last stage of the state processing, when all checking is completed.

The set D is used to record which quantifier nodes need to have values

Algorithm 1 Procedure check()

```
for all $s \in S$ in the trace order do
  $D = \emptyset$
  for all $e(\bar{v}) \in L_E(s)$ do
    for all $f \in nodes_e$ do
      update values stored in $f$
      for all $f'$ such that $f \xrightarrow{i} f'$ is a creation edge do
        for all $I \in \mathcal{I}_{f'}(I_e)$ do
          if $v_i \notin Ind_I(f')$ then
            add $v_i$ to $Ind_I(f')$
            initialize $nodes_{f'}$ at $I$
      for all $f'$ such that $f \rightarrow f'$ is an update edge do
        for all $I \in \mathcal{I}_{f'}(I)$ do
          add $(f', idx_e)$ to $A_{h(f')}$
  for all $h = 0, \ldots, h_{max}$, where $h_{max}$ is the maximum height of a node do
    for all $(f, I) \in A_h$ do
      for all $I_f \in \mathcal{I}_f(I)$ do
        $val_t := compute_{I_f}(f)$
        if $val_t \neq val_f(I_f)$ then
          $val_f(I_f) := val_t$
          for all $f \rightarrow f'$ do
            add $(f', I_f)$ to $A_{h(f')}$
            if $f \rightarrow f'$ is a destruction edge then
              add $(f', I_f)$ to $D$
  for all $(f, I) \in D$ do
    for all $I' \in \mathcal{I}_f(I)$ do
      remove $I(x)$ from $Ind_{I'}(f)$
      for all $f' \in nodes_f$ do
        remove $val_{f'}(I')$
```

removed from their index sets as a result of destruction events that happened in the course of evaluation of the current state. Let (f, I) be an element of D. The set of valuations for which the destruction event applies is $\mathcal{I}_f(I)$. If x is the variable that f binds, then $I(x)$ is the value that needs to be removed from every index set $Ind_{I'}(f), I' \in \mathcal{I}_f(I)$.

Complexity. We first address the question of space needed for checking of a given formula f over a trace S when the checker has reached $s \in S$. As in the propositional case, states of S are not stored directly. However, the amount of information stored in the graph depends on the number of creation events in the trace. This amount of information is the total number of entries in the value maps of the nodes. Each entry has a constant size (modulo representation of time values). Consider a node in the graph with free variables $x_1, \ldots, x_n$. The size of the value map stored in the node is proportional to the product of the index sets $I_{x_1}, \ldots, I_{x_n}$, where I_{x_i} is the largest of the index sets stored in the quantifier node where x_i is bound. The size of each such index set is proportional to $|S|$, which is measured as the number of states in S, whose timestamps do not exceed $\tau(s)$. The number of free variables in a node is bounded by the *quantifier nesting depth* k of f. Stated in terms of the graph representation of f, it is the maximum number of quantifier nodes along the path from any node in the graph to the node for f. Although k can be proportional to $|f|$ (measured in the number of nodes in the graph), practically it is much smaller. Finally, the number of nodes in the graph is exactly $|f|$. As a result, the required space is $O(|f| \cdot |S|^k)$.

A similar argument allows us to establish the time complexity of the algorithm. Here we note that updates to the values of nodes are performed in the order of increasing node heights, and thus each value in the value map stored within a node can be changed exactly once during processing of a state. Therefore, the total number of entries into the sets A_h cannot exceed the size of the value maps shown above. Processing of each entry takes constant time.

5 Conclusions

We presented a logic for specifying dynamic properties of systems for run-time monitoring and checking. Quantifiers of the logic allow us to bind values of attributes in observed events to be bound to variables in the formulas. Depending on the interpretation of the event attributes, the values can be used as data values for evaluation of predicates or as indices into dynamically changing index sets. This feature allows us to specify and check properties of systems comprised of dynamically created and destroyed objects.

Future work. We are implementing the checking algorithm for LC_ν in the MaC toolset. MEDL, the language of MaC, is a specification language based on LC_p. We will extend MEDL with the features of LC_ν. In this extension, we will investigate a balance between expressive power and efficiency of checking. The complexity of LC_ν checking is exponential in the quantifier nesting depth, while most properties we have come across in practice exhibit a very low nesting depth. We will look for natural ways of restricting the nesting depth, and also make specifications easier to write.

Acknowledgments. The motivation for LC_ν was gained in a case study performed within the DARPA PCA program. We wish to thank Mohammed Amduka and Michael Junod from the Lockheed Martin Advanced Technology Lab for insightful discussions of the system requirements.

References

[1] M. Abadi. The power of temporal proofs. *Theoretical Computer Science*, 65(1):35–83, 1989.

[2] R. Alur and T.A. Henzinger. A really temporal logic. *Journal of the ACM*, 41(1):181–204, January 1994.

[3] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *Proceedings of 5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI 2004)*, volume 2937 of *LNCS*, pages 44–57, January 2004.

[4] N. Bjørner, A. Browne, M. Colón, B. Finkbeiner, Z. Manna, H. Sipma, and T. E. Uribe. Verifying temporal properties of reactive systems: A step tutorial. *Formal Methods in System Design*, 16(3):227–270, 2000.

[5] S.M. Cho, H.K. Kim, S.D. Cha, and D.H. Bae. A semantics of sequence diagrams. *Information Processing Letters*, 84:125–130, 2002.

[6] D. Distefano, J.-P. Katoen, and A. Rensink. On a temporal logic for object-based systems. In *Formal Methods for Open Object-Based Distributed Systems (FMOODS-2K)*, pages 305–326, 2004.

[7] D. Distefano, A. Rensink, and J.-P. Katoen. Model checking birth and death. In *Foundations of Information Technology in the Era of Network and Mobile Computing*, volume 223 of *IFIP Conference Proceedings*, pages 435–447, 2002.

[8] K. Havelund and G. Rosu. Monitoring Java programs with JavaPathExplorer. In *Proceedings of the Workshop on Runtime Verification*, volume 55 of *Electronic Notes in Theoretical Computer Science*. Elsevier Publishing, 2001.

[9] T.A. Henzinger. Half-order modal logic: How to prove real-time properties. In *Proceedings of PODC '90*, pages 281–296, 1990.

[10] M. Kim, S. Kannan, I. Lee, O. Sokolsky, and M. Viswanathan. Java-MaC: a run-time assurance approach for Java programs. *Formal Methods in Systems Design*, 24(2):129–155, March 2004.

[11] I. Lee, S. Kannan, M. Kim, O. Sokolsky, and M.Viswanathan. Runtime assurance based on formal specifications. In *Proceedings of the Int. Conf. on Parallel and Distributed Processing Techniques and Applications - PDPTA'99*, June 1999.

[12] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *Journal of the ACM*, 32(3):733–749, 1985.

[13] A.P Sistla and O. Wolfson. Temporal triggers in active databases. *IEEE Transactions on Knowledge and Data Engineering*, 7(3), June 1995.

[14] E. Yahav, T. Reps, M. Sagiv, and R. Wilhelm. Verifying temporal heap properties specified via evolution logic. In *Proc. of the 12th European Symposium on Programming, ESOP 2003*, volume 2618 of *LNCS*, April 2003.